Cika Electrónica

Nota de Aplicación: CAN-035
Título: **Utilización de displays LCD color con controladores S1D13706 y Rabbit**
Autor: Sergio R. Caprile, Senior Engineer

Revisiones	Fecha	Comentarios
0	27/06/05	

En la CAN-031 les presentamos los nuevos displays color LCD de 320x240. Dijimos que se trata de displays sin controlador, por lo que su utilización dentro de un sistema requiere de un controlador externo, o de la generación de todas las señales de control por parte del usuario. Veremos en esta nota la forma de utilizar un controlador de Epson, el S1D13706, para que se encargue de todo lo relacionado con el manejo del display, mientras que nosotros nos limitaremos a decirle cómo lo tiene que hacer, y darle la información a mostrar.

Breve descripción del S1D13706

Hardware

El S1D13706 es un controlador inteligente para displays LCD de alta resolución, permitiendo no sólo displays color pasivos sino también blanco y negro y TFT. El S1D13706 se encarga de generar las señales que necesita el display, como viéramos en la CAN-031, tanto para este tipo de display como para los otros mencionados, según como esté configurado. La configuración de este dispositivo es algo compleja, pero afortunadamente el fabricante nos provee un software que nos permite, a partir de los valores operacionales del display, generar los datos a escribir en los diferentes registros. La imagen a enviar al display se aloja en una RAM interna de 80KB (81920 bytes), la cual es direccionada de forma lineal por diecisiete líneas de address, y accesible mediante un bus de datos de 16-bits. En un espacio de direccionamiento de 8-bits adicional al de memoria, se encuentran los registros de control que nos permiten controlar el funcionamiento a voluntad. La interfaz entre el S1D13706 y el procesador host puede elegirse entre una variedad de procesadores específicos soportados, y dos modos genéricos (uno de los cuales emplearemos en la presente aplicación), mediante pines destinados a tal fin. El arbitraje de la memoria entre la generación de la imagen en el display y el acceso por parte del procesador se resuelve mediante la petición de *wait-states* al procesador; sin embargo, este controlador de avanzada presenta un tiempo garantizado máximo, lo que permite funcionar sin *wait-states* si se garantiza una mínima duración al ciclo de acceso. Funciona con procesadores y displays de 3,3 V, mientras que el core puede funcionar a 3,3V o a tensiones más bajas

Software

El S1D13706 se encarga de todo lo referente al despliegue de la imagen, la cual reside como dijéramos en su memoria interna (80KB). Mediante los registros de control, es posible indicar en qué zona de memoria comienza la pantalla y su tamaño, así como también la cantidad de bits asignados a definir el color de cada pixel, lo cual definirá la distribución de la memoria. Para 8bpp, cada byte representa un pixel y los mismos se distribuyen de arriba a abajo y de izquierda a derecha, conforme avanzan las posiciones de memoria. Para 4bpp, es similar, pero se empaquetan dos pixels por byte y el pixel de la izquierda ocupa el nibble más significativo. Es posible almacenar varias pantallas en memoria y cambiar la posición de inicio, siempre y cuando, claro está, su tamaño y resolución permitan que quepa. Existen además algunas funciones de soporte adicionales como picture-in-picture, las cuales no veremos en esta nota de aplicación en particular.

La cantidad de colores determinada es un índice dentro de una paleta RGB 6:6:6, es decir, 256K (262144) colores posibles, y el controlador maneja todo esto de forma transparente, el programador especifica el modo de operación, carga la paleta, y trabaja con los índices, similar a como se haría con cualquier pantalla o formato de archivo en modo indexado (indexed-color). Existe además un modo de 16bpp sin paleta, pero como 16bpp no permite una resolución de 320x240, no lo consideraremos

Desarrollo propuesto

Para mantener la simpleza, optamos por mantenernos dentro de un mapa de 64K para el direccionamiento de memoria. Esto limita la capacidad de almacenamiento interno de pantallas, pero permite simplificar el software de bajo nivel, albergando una pantalla completa de 320x240 pixels en 4bpp (16 colores por pixel).

Al direccionar registros, operamos directamente sobre el pin $M/\overline{R}$, que controla dicha función.

Algoritmos

Para ubicar un punto en pantalla, calculamos su posición en memoria sabiendo que alojamos dos pixels por byte, es decir: $mem=\frac{x}{2} + 160 \cdot y$. El resto de $\frac{x}{2}$ nos dirá qué nibble utilizar

Para graficar funciones, debemos tener en cuenta que la coordenada (0;0) se halla en el extremo superior izquierdo de la pantalla.

Para mostrar pantallas, deberemos agrupar los datos de modo tal de poder enviarlos de forma que aproveche de manera eficiente la estructura de memoria. Si comparamos la estructura de memoria del display con la forma de guardar imágenes en 16 colores en formato BMP, veríamos que son muy similares, por ejemplo: BMP va de abajo a arriba y el display de arriba a abajo, por lo que la imagen se ve espejada verticalmente. Además, BMP incluye un encabezado que contiene la paleta de colores.

Por consiguiente, para adaptar una imagen, debemos llevarla a la resolución deseada, reducirla a 16 colores, espejarla verticalmente, salvarla en formato BMP y por último descartar los 118 bytes del comienzo con algún editor hexa. Entre los bytes a descartar tomaremos los bytes 54 a 117, los cuales corresponden a la paleta en formato BGR0 (4 bytes), y la guardaremos como RGB.

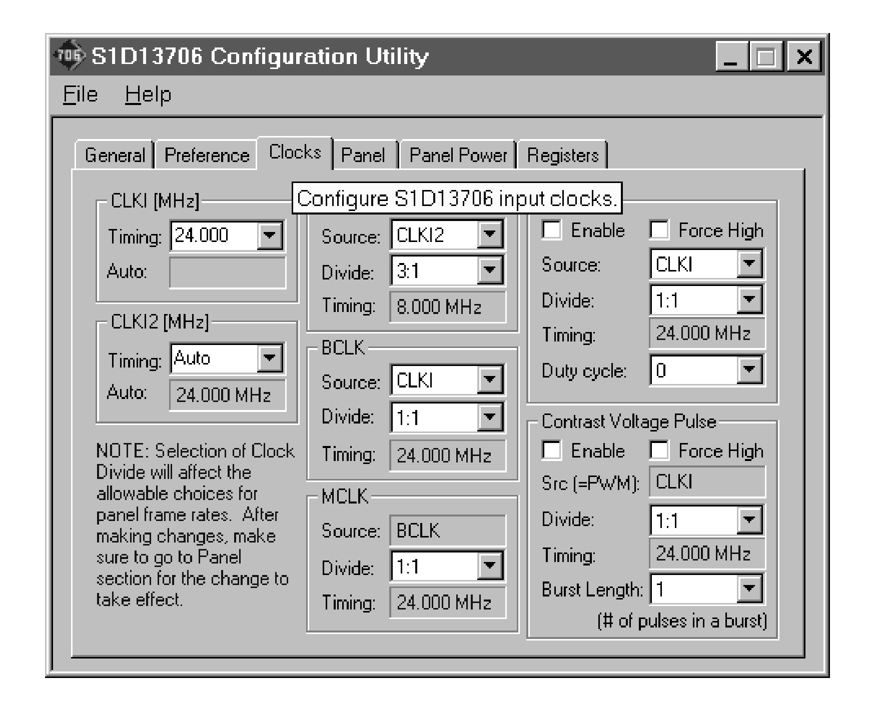
Para desplegar textos, deberemos generar las letras manualmente. La forma más común (y bastante eficiente) de almacenar tipografías en memoria, consiste en agrupar los pixels "pintados" del caracter en bytes, en el sentido horizontal, es decir, un byte aloja ocho pixels que corresponden a la parte superior del caracter, de izquierda a derecha, de MSB a LSB. Si el ancho del caracter es mayor a dieciséis pixels, entonces se utilizarán grupos de dos bytes. Ésta es la forma en la que se alojan los fonts provistos con las libraries de Dynamic C, y con un simple algoritmo los podemos convertir para su uso con un controlador de displays color. Simplemente, chequearemos el estado de cada pixel, y si éste está pintado, lo coloreamos en el display. De igual modo, si no lo está, podemos utilizar un color de fondo, o dejarlo sin modificar.

Configuración del S1D13706

Para obtener los valores a setear en cada uno de los registros, utilizamos el software de configuración provisto por el fabricante, definiendo un **display genérico color de un sólo panel STN, de 320x240 pixels, con datos en formato 2**. La frecuencia de reloj corresponderá al utilizado, en este caso 24 MHz, el cual haremos que se divida por tres, para mantener las frecuencias de operación dentro de los rangos recomendados por el fabricante del display (solapa *clocks*). En este desarrollo, utilizamos la entrada CLKI para ingresar el reloj.

Recordemos indicar 4bpp en la solapa *Preferences* El resto de los parámetros podemos dejar los valores por defecto, o leer detenidamente el manual técnico del S1D13706 para saber cómo configurar a nuestro antojo. Exportamos luego los datos en un archivo de tipo *C header file* (s1d13706.h), el cual podremos editar e incluir en el código para Rabbit.

Puede observarse un ejemplo del seteo de estas opciones en las pantallas que figuran en la hoja siguiente:



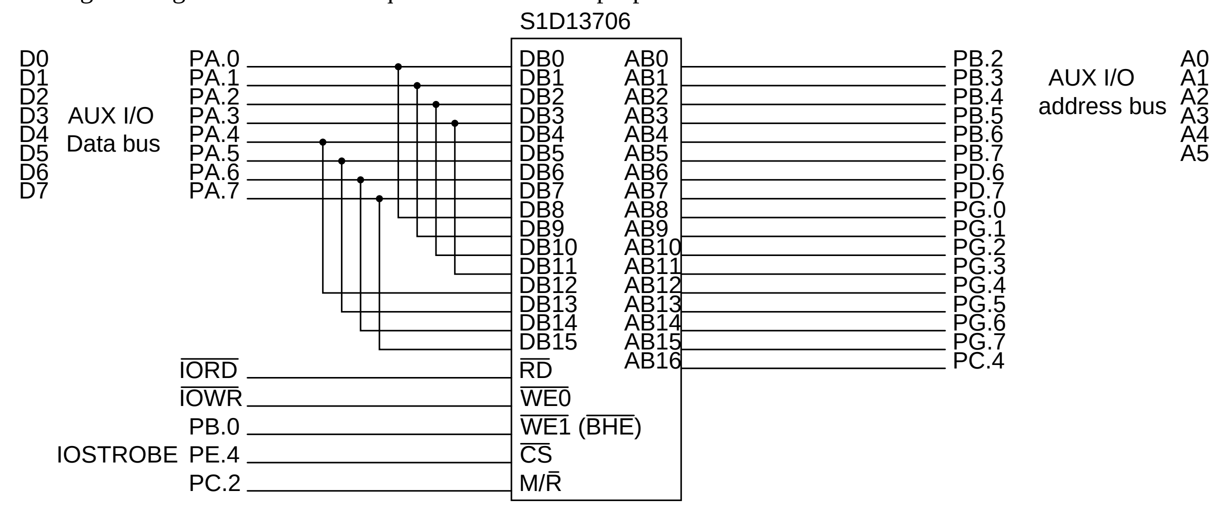
Hardware de interfaz

Debido a que el S1D13706 puede funcionar a 3,3 V o menos, hemos elegido para esta nota un RCM3300. Podemos emplear el bus de Rabbit para agilizar la operación, dado que Rabbit no tiene línea de $\overline{WAIT}$, *READY*, o equivalente, sino que genera una inserción automática de wait-states por configuración, configuraremos la cantidad de ciclos de espera necesarios para garantizar un tiempo de acceso mínimo que sea mayor al especificado para el controlador; siempre será más rápido que interrogar manualmente la línea de $\overline{WAIT}$. Debido a que no todo el bus de direcciones está disponible en los módulos (sí en el procesador para quien quiere desarrollar su propio hardware), generaremos parte de las direcciones pertinentes mediante I/O. En cuanto al bus de 16-bits, lo que haremos es duplicar nuestros 8-bits conectando parte alta y parte baja del bus entre sí, elegimos para esta nota el modo de operación "Generic #2", y le indicamos al controlador si la

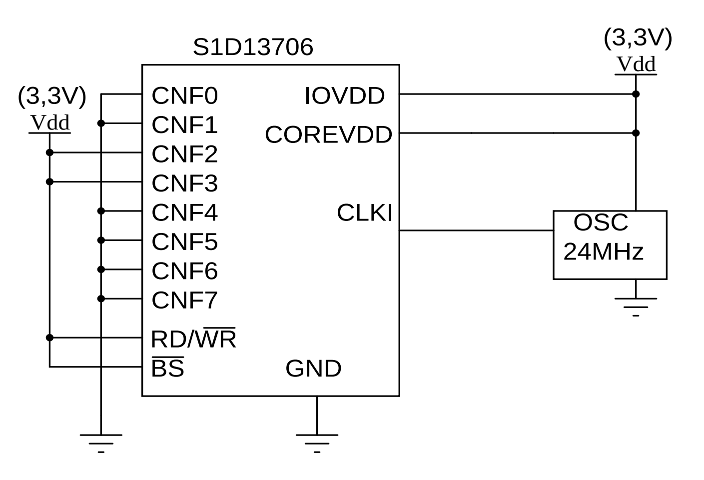
operación es en la parte baja o la parte alta del bus (a la usanza del 8086). De este modo, la información siempre sale del Rabbit por los mismos ocho pines, y el controlador la lee por los ocho altos o los ocho bajos, según indique la señal $\overline{BHE}$ (Bus High-byte Enable), que no es otra cosa que la inversión de A0. Para poder acceder al bus de Rabbit, deberemos utilizar el bus auxiliar de I/O que provee R3000. Para la frecuencia de reloj utilizada en esta nota, en un RCM3300, deberemos insertar unos 15 ciclos de espera en el rango de I/O externo a utilizar. Dado que empleamos PE.4, éste será de 0x8000 a 0x9FFF; seis de los bits menos significativos del bus de direcciones están accesibles en PB.2 a PB.7, mientras que PA.0 a PA.7 ofician de bus de datos.

Entre las limitaciones de cantidad de pines y posibilidad de operación, elegimos aquéllos que nos agilizaban las rutinas de manejo, sin interferir con los periféricos adicionales presentes en el kit de desarrollo utilizado (RCM3360). Queda como tarea para el lector, el estudio de la posibilidad de realizarlo con otros pines, emplear algún otro de los modos de interfaz del S1D13706, e incluso desarrollar un hardware con mapeo directo en memoria. Para los alcances de esta Nota de Aplicación, la performance obtenida es satisfactoria.

El diagrama siguiente indica el esquema de conexión propuesto:

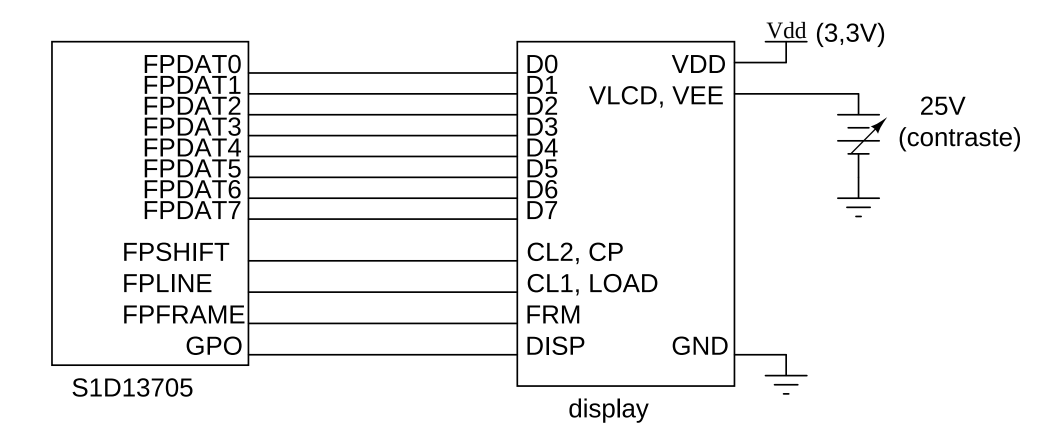


La selección del tipo de interfaz se realiza mediante los pines *CNFx*. El diagrama de configuración, alimentación, y misceláneos, es el siguiente:



Hardware de display

La conexión al display se realiza mediante las líneas analizadas en la CAN-031, como indica el diagrama a continuación:



La tensión de contraste es conveniente que sea variable para poder ajustarlo, podemos utilizar un step-up como el MC34063A (comercializado por Cika) y controlarla mediante un preset a modo de divisor resistivo, o proveerla de forma externa.

Software de bajo nivel

Tenemos un bus de 16-bits simulado dentro de uno de 8-bits, con señalización similar a la utilizada por el 8086, es decir, cuando operamos sobre el byte "alto" (parte "alta" del bus, D8 a D15), lo indicamos activando la señal $\overline{BHE}$. Nuestro bus de direcciones posee una parte baja "real" y una parte alta simulada mediante I/O, por lo que luego de realizar la simulación, procederemos a realizar una operación de I/O externo normal. A los fines prácticos incluimos toda la operación de generación de líneas de direcciones en una subrutina. Dado que tendremos bastantes datos para escribir, necesitamos hacerlo rápido, razón por la cual utilizamos assembler para las rutinas críticas.

```
#asm root
read13706::
      call addressbus              ; pone addresses y BHE, devuelve address I/O en DE
      ex de,hl
      ioe ld a,(hl)                ; lee port paralelo
      ld h,0
      ld l,a
      ret

write13706::
      call addressbus              ; pone addresses y BHE, devuelve address I/O en DE
      ld hl,(sp+4)                 ; lee dato
      ld a,l
      ex de,hl
      ioe ld (HL),a                ; pone dato
      ret

addressbus:
        ld hl,(sp+4)               ; lee address
        ex de,hl                   ; en DE
ab2:    ld hl,PGDR                 ; apunta a port
        ioi ld (hl),d              ; pone parte alta
        ld hl,PDDR                 ; apunta a port
        ioi ld (hl),e              ; pone parte baja (7,6)
        ld a,e                     ; A= A7-A0
        ld hl,PBDR                 ; apunta a port paralelo
        ioi set 0,(hl)             ; BHE=1
        rrc e                      ; debo activar BHE ? (test LSB=1)
        jr nc,ok1                  ; no
        ioi res 0,(HL)             ; sí, BHE=0
ok1:
        and 0x3F                   ; A6,A7 = 0
        ld e,a
        ld d,0x80                  ; I/O = 0x8000 + A5-A0 = 10000000 00xxxxxx
        ret
#endasm
```

Para poder observar algo, deberemos definir los colores, cargando la paleta. Eso lo hacemos mediante la siguiente rutina:

```
void writepalette(triplet *address)
{
int i;
unsigned char *ptr;

        ptr=&(address->RGB[0]);
        BitWrPortI(PCDR,&PCDRShadow,0,2);                          // M/R=0 => registers
        for(i=0;i<16;i++){
          write13706(0x0A,*ptr++);
          write13706(0x09,*ptr++);
          write13706(0x08,*ptr++);
          write13706(0x0B,i);
        }
        BitWrPortI(PCDR,&PCDRShadow,1,2);                          // M/R=1 => memory
}
```

Cada paleta estará definida como un array de tres elementos, o triplet, definiendo los valores correspondientes a R, G, y B. Si bien el controlador utiliza sólo 6-bits, definimos un byte para cada color por simplicidad y semejanza con VGA.

```
typedef union {
        unsigned char RGB[3];
} triplet;

const triplet VGApalette[16]= {
{       0x00, 0x00, 0x00},            /* BLACK  */
{       0x00, 0x00, 0xA0},            /* BLUE   */
{       0x00, 0xA0, 0x00},            /* GREEN  */
{       0x00, 0xA0, 0xA0},            /* CYAN   */
{       0xA0, 0x00, 0x00},            /* RED    */
{       0xA0, 0x00, 0xA0},            /* PURPLE */
{       0xA0, 0xA0, 0x00},            /* YELLOW */
{       0xA0, 0xA0, 0xA0},            /* WHITE  */
{       0x00, 0x00, 0x00},            /* BLACK  */
{       0x00, 0x00, 0xF0},            /* LT BLUE   */
{       0x00, 0xF0, 0x00},            /* LT GREEN  */
{       0x00, 0xF0, 0xF0},            /* LT CYAN   */
{       0xF0, 0x00, 0x00},            /* LT RED    */
{       0xF0, 0x00, 0xF0},            /* LT PURPLE */
{       0xF0, 0xF0, 0x00},            /* LT YELLOW */
{       0xF0, 0xF0, 0xF0}             /* LT WHITE  */
};
```

A continuación, la inicialización del chip. Los valores los obtuvimos utilizando el software de configuración provisto por el fabricante, según comentáramos.

```
typedef unsigned short S1D_INDEX;
typedef unsigned char  S1D_VALUE;

typedef struct
{
    S1D_INDEX Index;
    S1D_VALUE Value;
} S1D_REGS;

const static S1D_REGS aS1DRegs[] =
{
    {0x04,0x00},   // BUSCLK MEMCLK Config Register
    {0x05,0x22},   // PCLK Config  Register
    {0x10,0xD0},   // PANEL Type Register
    {0x11,0x00},   // MOD Rate Register
    {0x12,0x2B},   // Horizontal Total Register
    {0x14,0x27},   // Horizontal Display Period Register
    {0x16,0x00},   // Horizontal Display Period Start Pos Register 0
    {0x17,0x00},   // Horizontal Display Period Start Pos Register 1
    {0x18,0xFA},   // Vertical Total Register 0
    {0x19,0x00},   // Vertical Total Register 1
    {0x1C,0xEF},   // Vertical Display Period Register 0
```

```
    {0x1D,0x00},   // Vertical Display Period Register 1
    {0x1E,0x00},   // Vertical Display Period Start Pos Register 0
    {0x1F,0x00},   // Vertical Display Period Start Pos Register 1
    {0x20,0x87},   // Horizontal Sync Pulse Width Register
    {0x22,0x00},   // Horizontal Sync Pulse Start Pos Register 0
    {0x23,0x00},   // Horizontal Sync Pulse Start Pos Register 1
    {0x24,0x80},   // Vertical Sync Pulse Width Register
    {0x26,0x01},   // Vertical Sync Pulse Start Pos Register 0
    {0x27,0x00},   // Vertical Sync Pulse Start Pos Register 1
    {0x70,0x02},   // Display Mode Register
    {0x71,0x00},   // Special Effects Register
    {0x74,0x00},   // Main Window Display Start Address Register 0
    {0x75,0x00},   // Main Window Display Start Address Register 1
    {0x76,0x00},   // Main Window Display Start Address Register 2
    {0x78,0x28},   // Main Window Address Offset Register 0
    {0x79,0x00},   // Main Window Address Offset Register 1
    {0x7C,0x00},   // Sub Window Display Start Address Register 0
    {0x7D,0x00},   // Sub Window Display Start Address Register 1
    {0x7E,0x00},   // Sub Window Display Start Address Register 2
    {0x80,0x50},   // Sub Window Address Offset Register 0
    {0x81,0x00},   // Sub Window Address Offset Register 1
    {0x84,0x00},   // Sub Window X Start Pos Register 0
    {0x85,0x00},   // Sub Window X Start Pos Register 1
    {0x88,0x00},   // Sub Window Y Start Pos Register 0
    {0x89,0x00},   // Sub Window Y Start Pos Register 1
    {0x8C,0x4F},   // Sub Window X End Pos Register 0
    {0x8D,0x00},   // Sub Window X End Pos Register 1
    {0x90,0xEF},   // Sub Window Y End Pos Register 0
    {0x91,0x00},   // Sub Window Y End Pos Register 1
    {0xA0,0x00},   // Power Save Config Register
    {0xA1,0x00},   // CPU Access Control Register
    {0xA2,0x00},   // Software Reset Register
    {0xA3,0x00},   // BIG Endian Support Register
    {0xA4,0x00},   // Scratch Pad Register 0
    {0xA5,0x00},   // Scratch Pad Register 1
    {0xA8,0x00},   // GPIO Config Register 0
    {0xA9,0x80},   // GPIO Config Register 1
    {0xAC,0x00},   // GPIO Status Control Register 0
    {0xAD,0x00},   // GPIO Status Control Register 1
    {0xB0,0x00},   // PWM CV Clock Control Register
    {0xB1,0x00},   // PWM CV Clock Config Register
    {0xB2,0x00},   // CV Clock Burst Length Register
    {0xB3,0x00},   // PWM Clock Duty Cycle Register
};

#define S1DNUMREGS 54
#define S1DMEMSIZE 38400

void init13706()
{
int i;

       BitWrPortI(PCDR,&PCDRShadow,0,2);                    // M/R=0 => registros
       for(i=1;i<S1DNUMREGS;i++)
       write13706((aS1DRegs[i].Index),aS1DRegs[i].Value);
       writepalette(VGApalette);                            // al regresar, M/R=1 => memoria
}
```

Software

El resto del software lo escribimos mayormente en C, por comodidad y velocidad de desarrollo. Se trata de simples y comunes rutinas que no incluiremos aquí para no extender el texto, pero que el lector puede obtener del archivo adjunto con el software, o consultar en cualquiera de las otras notas de aplicación, dado que son muy similares.

Una excepción es el volcado de imágenes, para el cual escribimos una función especial en assembler. Dado el tamaño de una imagen ($\frac{320\times240}{2}=38400\,bytes$), necesitamos una función rápida que copie la imagen directamente desde xmem, leyendo por words y escribiendo de a dos bytes en el S1D13706. Básicamente, tomamos la idea de *xmem2root()* y lo adaptamos a nuestros propósitos, por simplicidad usamos una función externa que resuelve el mapeo de dirección absoluta en 20-bits a valores de *XPC* y dirección lógica.

```
#asm root

;IY: dirección en el display
;(IX+0): dato pixel ixquierdo
;(IX+1): dato pixel derecho
writew13706::
      ld hl,iy                              ; get A15-A0
      ex de,hl                              ; in DE
      call ab2                              ; put addresses and BHE
      ld a,(IX+0)                           ; lee dato
      ex de,hl
      ioe ld (HL),a                         ; pone dato
      ld hl,iy                              ; lee address
      inc hl                                ; inc address
      ex de,hl                              ; A15-A0 in DE
      call ab2                              ; address bus
      ex de,hl
      ld a,(IX+1)                           ; lee dato
      ioe ld (HL),a                         ; pone dato
      ret


xdumpblk::
       ld a,xpc                             ; salva frame pointer, y xpc
       push    af
       push    ix

       ld hl,(sp+10)                        ; Obtiene xmem address (long) en BC:DE
       ld c,l
       ld b,h
       ld hl,(sp+8)
       ex de,hl
                               ; Convierte BC:DE a XPC + address
       call LongToXaddr                     ; DE = logical address, A = xpc
       ld xpc,a
       ex de,hl
       ld ix,hl                             ; IX = source


       ld iy,(sp+6)                         ; iy=dest, hl=length
       ld hl,(sp+12)

       ld c,h                               ; c:b=length
       ld b,l
       ld a,l
       or a
       jr nz,xcbf_loop
       dec c

xcbf_loop:
       call writew13706                     ; preserva BC
       ld de,0x0002
       add ix,de
       add iy,de
       djnz xcbf_loop
       xor a
       dec c
       jp p,xcbf_loop

xcbf_done:
       pop ix
       pop af
       ld xpc,a
       ret
#endasm

root useix void LCD_dump(long imgdata, triplet *paldata)
{

unsigned int dest,tocopy,len;

       len=S1DMEMSIZE/2;
       dest=0;
```

```
        imgdata+=sizeof(long);
        writepalette(paldata);
        while(len) {
                tocopy=2048-(dest&2047);
                if(tocopy>len)
                        tocopy=len;
                xdumpblk(dest,imgdata,tocopy);
                dest+=(tocopy<<1);
                imgdata+=(tocopy<<1);
                len-=tocopy;
        }
}
```

El algoritmo de generación de textos en 4bpp podría optimizarse en assembler, sin embargo hemos decidido dejarlo en C para los alcances de esta nota.

Aprovechando que las fonts de Dynamic C están definidas como libraries, las podemos incluir automáticamente en xmem. Definiremos una simple estructura para guardar algunos parámetros de cada tipografía que nos permitan acelerar su impresión, estos datos los obtenemos observando el archivo que contiene la tipografía, que no es otra cosa que código fuente:

```
#use "6X8L.lib"
#use "12X16L.lib"

typedef struct {
        unsigned long *font;
        unsigned char  lpc;             // líneas por character
        unsigned char  Bpcl;            // bytes por línea de caracter (1 ó 2)
        unsigned char  bpcl;            // bits por línea de caracter
} FontInfo;

const static FontInfo fontinfo[]={
{&Font6x8,8,1,6},
{&Font12x16,16,2,12}
};
```

A continuación, veremos una rutina para imprimir un caracter en 4bpp:

```
void LCD_putchr ( int font, unsigned long daddress, char chr , int color, int bcolor)
{
int i,j,ii,jj;
unsigned long address,dispaddress;
int data,ddata,aux;

        i=fontinfo[font].lpc;                                   // líneas por character
        ii=fontinfo[font].Bpcl;                                 // bytes por línea de caracter
        jj=fontinfo[font].bpcl;                                 // bits por línea de caracter
        address=*(fontinfo[font].font)+i*ii*(chr-0x20);         // ubica caracter en tipografía
        dispaddress=daddress;                                   // apunta a display
        while(i--){
                data=xgetint(address);
                aux=(data&0xFF)<<8;                             // swap bytes (MSB - LSB)
                data=((data>>8)&0xFF)+aux;
                address+=ii;
                j=jj;
                while(j--){
                        ddata=read13705((unsigned int)(dispaddress/2)); // lee display
                        aux=-1;
                        if(data&0x8000)                         // chequea si pinta
                                aux=color;
                        else
                                if(bcolor>=0)                   // bcolor =-1 => transparente
                                        aux=bcolor;
                        if(aux != -1){
                                if(dispaddress&1){              // ubica nibble
                                        ddata&=0xF0;            // borra bits a escribir
                                        ddata|=aux;             // setea color del pixel derecho
                                }
                                else {
                                      ddata&=0x0F;
                                      ddata|=(aux<<4);          // setea color del izquierdo
                                }
                        }
```

```
                        write13705((unsigned int)(dispaddress/2),ddata);
                        data<<=1;
                        dispaddress++;
                }
                dispaddress=daddress+=320;                      // línea siguiente
        }
}
```

El cálculo de la dirección inicial en pantalla para cada caracter de un string, la realizamos de la siguiente forma:

```
void LCD_printat (int font,unsigned int row,unsigned int col,char *ptr,int color,int bcolor)
{
unsigned long address;

        address=320L*row;                                       // ubica dirección en display
        do {
                address+=col;
                LCD_putchr (font,address,*ptr++,color,bcolor);          // pone caracter
                col=fontinfo[font].bpcl;                                // siguiente
        }       while (*ptr);
}
```

Para escribir un texto, simplemente llamamos a esta rutina, teniendo cuidado de no excedernos en los límites útiles:

```
        LCD_printat(0,20,20,"Cika Electronica",15,6);
```

Finalmente, como punto importante, tengamos en cuenta al inicializar el módulo de setear correctamente los pines bidireccionales en el sentido en que los usamos, y todos en el estado inactivo. Inmediatamente después, inicializamos el chip:

```
        WrPortI ( PGDDR,&PGDDRShadow,'\B11111111' );
        WrPortI ( PEDDR,&PEDDRShadow,'\B00010000' );
        WrPortI ( PDDDR,&PDDDRShadow,'\B11000000' );
        WrPortI ( PBDDR,&PBDDRShadow,'\B11111111' );
        WrPortI ( PBDR,&PBDRShadow,'\B11000000' );
        WrPortI ( PCDR,&PCDRShadow,'\B11101111' );              // AB16=0, M/R = M

// Use Port E bit 4 for Chip Select with 15 wait-states
#define STROBE          0x10
#define CSREGISTER      IB4CR
#define CSSHADOW        IB4CRShadow
#define CSCONFIG        0x08

        // Initialize Port E bit to be a normal I/O pin
        WrPortI(PEFR, &PEFRShadow, (PEFRShadow|STROBE));

        // Initialize Port E bit to be an output pin
        WrPortI(PEDDR, &PEDDRShadow, (PEDDRShadow|STROBE));

        // Initialize Port E bit to be a chip select.
        WrPortI(CSREGISTER, &CSSHADOW, CSCONFIG);

        // Set Port E bit to be clocked by PCLK/2
        WrPortI(PECR, &PECRShadow, (PECRShadow & ~0xFF));

        init13706();
```

Nota importante

Tanto el fabricante del display como el del controlador recomiendan respetar un ciclo de encendido y apagado para maximizar la vida útil del display. En el caso del S1D13706, ésto se hace controlando manualmente la señal GPO, por lo que deberemos, luego de inicializado el display, esperar un cierto tiempo y luego habilitar el display. Antes de apagar haremos el proceso inverso, deshabilitando el display antes de inhibir las señales de control y retirarle la alimentación.

```
        BitWrPortI(PCDR,&PCDRShadow,0,2);                       // M/R=0 => registros
        MsDelay(1000);
```

```
write13706(0xAD,0x80);                                  // habilita display (GPO=1)
BitWrPortI(PCDR,&PCDRShadow,1,2);                       // M/R=1 => memoria
```